View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aaltodoc Publication Archive

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Lauri Luotola

Succeeding in a software rewrite project within a startup:

A case study

Master's Thesis
Espoo, November 16, 2017

Supervisor: Professor Kari Smolander
Advisor: Petri Avikainen M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Lauri Luotola		
Title:			
Succeeding in a software rewrite project within a startup: A case study			
Date:	November 16, 2017	**Pages:**	69
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Kari Smolander		
Advisor:	Petri Avikainen M.Sc. (Tech.)		

Software projects are notorious for their failure rates and software maintenance is a complex task that often becomes more time-consuming as the software ages. In modern software development, maintenance is often done in an iterative fashion with the help of continuous integration and deployment tools to help with quality assurance.

This thesis is a postmortem case study of the design and development involved in a user interface rewrite project conducted for a healthtech SaaS-product. The focus of the study is on investigating how efficient the methods of working were, what pain points were identified and how well the risks were managed for the project. It aims to provide insight on how early-stage companies with limited resources can see through a sizable effort such as this efficiently. Focus is also given to whether a transition towards a microservice-architecture is a viable choice within this context.

The key findings from the conducted case study are that even when following agile practices, a systematic approach to software engineering is essential for success. Projects should have a clear scope and clear responsibilities in order for their success to be measurable. Team composition and individual skills are the crucial elements in a development team, and tools and practices only strengthen the results of individuals. However, open communication, motivated individuals and visibility into progress are also essential.

Keywords:	rewrite, maintenance, risk management, technical debt, microservices
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Computer, Communication and Information Sciences

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Lauri Luotola		
Työn nimi:	Ohjelmistoprojektin riskien ja prosessien hallinta startup-yrityksessä: Tapaustutkimus		
Päiväys:	16. marraskuuta 2017	**Sivumäärä:**	69
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Kari Smolander		
Ohjaaja:	Diplomi-insinööri Petri Avikainen		

Ohjelmistoprojektien epäonnistuminen on tutkitusti yleistä ja ohjelmistojen ylläpito on kompleksinen tehtävä, jonka vaatimat resurssit usein kasvavat ohjelmiston vanhetessa. Modernissa ohjelmistokehityksessä ylläpito usein tehdään iteratiivisesti, hyödyntäen jatkuvaa integraatiota laadunvarmistuksen apuna.

Tämä diplomityö on tapaustutkimus terveysteknologiaan keskittyneen SaaS-sovelluksen uudistukseen liittyneestä kehitys- ja suunnittelutyöstä. Tutkielma keskittyy tutkimaan projektin riskinhallintaa, kehitysmetodien ja -prosessien tehokkuutta sekä löytämään näistä kipupisteitä. Työn tavoitteena on löytää resulsseiltaan rajallisille alkuvaiheen ohjelmistoyrityksille soveltuvia työtapoja sekä selvittää, kuinka tämänkaltainen laaja kehitystyö voidaan suorittaa onnistuneesti. Tutkimus myös pyrkii selvittämään onko mikropalveluarkkitehtuuriin siirtyminen kannattavaa tässä kontekstissa.

Työn tuloksena havaittiin, että systemaattinen lähestyminen ohjelmistokehitykseen on olennaista onnistumisen kannalta myös ketteriä menetelmiä hyödynnettäessä. Projekteilla tulisi olla selkeä laajuus ja selkeät tavoitteet, jotta projektin onnistumista voidaan mitata objektiivisesti. Kehitystiimin dynamiikka ja yksilöiden taidot ovat tärkein osa kehitystiimiä, ja työkaluilla ja menetelmillä on vain toissijainen vaikutus yksilöiden suorituskykyyn. Toisaalta myös avoin kommunikaatio, motivoituneet yksilöt ja kehityksen läpinäkyvyys ovat olennaisessa asemassa.

Asiasanat:	ylläpito, riskienhallinta, tekninen velka, mikropalvelut
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor Kari Smolander and advisor Petri Avikainen for their support and encouragement in getting this thesis done. Same goes for my friends and family; especially my mother for constantly reminding me to keep working, and Skipoli for the various broken bones and unforgettable memories. It sure did take a while, but here it is — and done right on time before Länsimetro (thanks to them for not hurrying with it)!

Helsinki, November 16, 2017

Lauri Luotola

Abbreviations and Acronyms

API	Application Program Interface
CI	Continuous Integration
MVC	Model-View-Controller
PROM	Patient Reported Outcome Measure
QA	Quality Assurance
REST	Representational State Transfer
SaaS	Software as a Service
SLA	Service-Level Agreement
SPA	Single-Page Application
SOA	Service Oriented Architectures
WIP	Work In Progress

Contents

Abbreviations and Acronyms **5**

1 Introduction **8**
1.1 Problem statement and study methods . . . 9
1.2 Structure of the Thesis . . . 10

2 Literature review **12**
2.1 Maintenance as a part of software development . . . 12
2.1.1 Technical and social debt . . . 13
2.1.2 Design smells and software maintainability . . . 14
2.2 Developer motivation and team dynamic in agile teams . . . 17
2.2.1 Developer experience . . . 17
2.2.2 Kanban as a development workflow . . . 18
2.3 Risk management and the cost of change . . . 19
2.3.1 Continuous delivery and -integration . . . 19
2.3.2 Microservice-architecture . . . 22

3 The Kaiku Health rewrite project **25**
3.1 Kaiku Health as a product . . . 25
3.2 Development workflow . . . 26
3.3 Release management . . . 27
3.4 The old architecture . . . 28
3.5 Objectives of the rewrite . . . 28
3.6 Design phase and risk assessment . . . 29
3.7 Development phase . . . 30
3.8 Rollout . . . 31

4 Research strategy **33**
4.1 Objective of the study . . . 33
4.2 Data collection techniques . . . 33
4.3 Interviewee selection . . . 35

4.4 Structure of the interviews 36

5 Empirical study: Kaiku Health user interface overhaul 38

5.1 Planning and risk analysis 38
5.2 The amount of waste 39
5.2.1 Partially done work and task switching 39
5.2.2 Defects 40
5.2.3 Extra processes 41
5.2.4 Waiting 41
5.3 Ways of working 42
5.3.1 Code review and continuous integration 42
5.3.2 Team dynamic 43
5.3.3 Tracking progress 44
5.4 Success of the project 45
5.4.1 Technical success 45
5.4.2 Transitioning to microservices 46
5.4.3 Essential factors in a successful launch 47
5.5 Summary of the interviews 48

6 Discussion 50

6.1 Risk assessment in a near-complete frontend rewrite 50
6.1.1 The importance of quantifiable goals 51
6.1.2 Architectural considerations and the suitability of microservices for a startup 52
6.2 How to succeed in a major development effort within a startup 54
6.2.1 Development processes 55
6.2.2 People 56
6.2.3 Project content 57
6.3 Methodological considerations 58

7 Conclusions 59

A Interview structure (Finnish) 67

Chapter 1

Introduction

In the recent years, the role of software in healthcare has increased, and patient-facing systems have started to gain adoption, with the goal of reducing costs and improving quality of care. Pioneers in the field built their products alongside their users, but in the healthcare domain this is often challenging, mainly due to heavy regulation [61]. This makes it difficult for software developers to build quality products that fit the unique processes of healthcare, especially when it comes to specialized areas such as cancer care. As a result of this, there is growing concern about the quality of the software systems produced for healthcare [61].

Kaiku Health (formerly known as Netmedi) is a healthtech company focused on offering a Software-as-a-Service (SaaS) -product, also called Kaiku Health. The product aims to accelerate the gathering of patient-reported outcome measures (PROMs). This is done by enabling real-time communication between patients and their clinicians, as well as providing detailed surveys and reporting for the clinicians to automatically pinpoint possible alarming conditions such as adverse effects to medication. The value proposition is that by improving the communication between the clinicians and patients, the quality of the treatment and quality of life of the patients also increases. In practice, this also decreases the amount of unnecessary visits to the clinics, which in turn enhances the efficiency of the clinicians.

The company started a brand reform in 2016 to freshen the appearance of both the company itself and its core product, Kaiku Health. Beginning the rebranding quickly led us to realize that taking up on such a large update of the product could also facilitate a move towards a more future-proof architecture. Rather than just update the visuals, it was decided that a partial rewrite of the product would be done. This consisted of updating a large part of the user interface to use new technologies and separating the new frontend from the monolith codebase. The aim was to ensure future maintainability

while improving the user experience of the product as a whole.

The subject of this thesis stems from this process and insights gained from the project; being responsible for a business-critical project with the limitations of a startup team has unique challenges that should be taken into account when considering such a task. The project itself was quite large in terms of work it required, and the difficulty was amplified by the small development team and limited resources available in the company.

The new user interface began its rollout to customers gradually in Q2 of 2017. This thesis studies the process from the design phase to delivery from the perspective of the team involved in the project. The aim is to discover possible areas where processes could be improved and what can be learned from taking on a large task such as this in a small sized company. Focus is on considering the risks involved and how team dynamic and ways of working affect the success of such projects. In addition to this, the technical details are considered in terms of the project's effect on technical debt and the architectural choices made, such as the feasibility of breaking up a large monolith codebase in order to begin a transition towards a microservice-architecture.

1.1 Problem statement and study methods

This thesis aims to answer the following questions:

- *RQ1: What risks are involved in a large rewrite project for a startup company?*
- *RQ2: How suitable are microservices for a startup company?*
- *RQ3: How to succeed in a large rewrite project and are they an efficient way to reduce technical debt?*

The interest towards the risk management angle originates from the risks we recognized and the the difficulties we faced while planning and implementing the frontend rewrite for Kaiku Health. While also technologically complex, I found the cultural challenges and project management of such a big task in a small company a more unique problem to assess. Software project success and team performance have been widely studied in the past, however research is lacking in the context of a small company and small development team.

Question 1 finds out whether the chosen way of implementation was ideal and how well it was thought out prior to beginning the development phase.

Question 2 stems from the initial idea that the rewrite would have been used to facilitate a move towards a microservice architecture. The goal is to find out whether microservices are a viable choice for a small startup and what are the benefits and disadvantages of breaking a monolithic architecture. It also focuses on the technical aspects of the implementation as well as how it affected development that went into the product as a whole. Focus is given to the importance of minimizing technical complexity in terms of developer productivity.

Question 3 assesses the success of the project as a whole and relates that to prior academic findings, aiming to find key elements to succeeding in a similar rewrite project. This has a large emphasis on the retrospective findings from the development team and how they perceived the success of the project. Focus is on the social aspects of software project success, such as ways of working, team dynamic and communication.

By gaining insight on the processes and workflows used in the development of this project, the aim of the case study is to find patterns that companies, especially ones with limited resources, could utilize in a similar situation. Therefore the literature study focuses mostly on software maintenance, risk management and process management area. The aim is to find what are common pitfalls and best practices in the context of agile development for small companies. These themes are also reiterated by conducting postmortem interviews with people involved in the case project.

1.2 Structure of the Thesis

This introduction is followed by Chapter 2 which includes extensive literature research on maintenance and risk management in software engineering. Emphasis is also given to project management aspects such as what motivates software developers, what affects productivity and how technical debt relates to software maintenance. The concept of a microservice architecture is studied and what benefits and disadvantages it could bring.

Chapter 3 gives background information on the case company environment and the technical and practical sides of the rewrite project. Reasoning is given on what led to the birth of the product and what its main focus areas are, giving a look into the healthcare domain the company operates in. The factors that led to the rewriting of the frontend codebase are discussed and background is given on the team composition and development workflow that was present at the company during the writing of this thesis. This includes an introduction to the state of Kaiku Health as a product prior to the rewrite project and also explains the architectural considerations and

decisions made to support the rewrite project. It also gives insight into the planning that went into the project and the rollout phase.

Chapter 4 discusses the methodology and practical arrangements used in this case study. Structure of the interviews is explained, as well as reasoning behind the chosen methods, subjects and themes.

Chapter 5 contains evaluation of the interview results. The conducted interviews are summarized and common topics and problem areas are identified and analyzed.

Chapter 6 discusses the results derived from the case study as well as the literature research. Focus is on finding answers for the research questions and seeing what could be learned from the project.

Finally, Chapter 7 concludes the thesis with a summary of the study's results and looks into possible future developments.

Chapter 2

Literature review

Looking into the success and reasoning behind Kaiku Health's rewrite scenario, it can be seen as a part of software maintenance. Therefore it is important to understand common aspects of maintenance in software projects in general, as well as what affects project success and how agile teams operate. As the initial idea was that the rewrite could have been used to facilitate a move towards a microservice architecture, background is also given on where the concept originates from and what are the benefits and disadvantages of breaking a monolithic architecture.

2.1 Maintenance as a part of software development

Maintenance is one of the greatest challenges for software developers, and maintainability is hard to quantify. Changes may be required to keep the software performing as intended, to add new features or to adjust to changed requirements. One of the objectives of a software product is to serve the needs of its users, and these needs can and often will change during the lifetime of the software. [29]

The IEEE defines software maintenance as the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [32]. Swanson [55] classified maintenance to three types:

- Corrective maintenance aims at fixing defects
- Adaptive maintenance is done to reflect changing requirements and environment

- Perfective maintenance improves the performance and maintainability of the product

Maintenance and risk management have both been widely studied, although the methods of both software engineering and delivering updates both have greatly evolved from the early days to recent times. A notable distinction between software engineering and other, more traditional branches of engineering is the shortage of well accepted metrics of software development. The methods of working vary greatly from company to company due to different team compositions, architectural decisions and business goals, so there often is no silver-bullet solution to a specific problem.

Maintenance is estimated to consume 40-75 percent of the software development effort [50, 59]. It is differentiated from other software development by the constraints of the existing system; posing limitations both in design and architecture. Defects in business applications can cause lost productivity, losses in revenue and possible legal risks in case of incorrect data [33]. Once the end users are familiar with the product, introducing major changes need extra carefulness to not break existing workflows or cause unnecessary overhead for the people that depend on the product.

2.1.1 Technical and social debt

Technical debt is a well-known concept advocated by the agile development community; it reflects the extra development work that arises when code quality decreases as the result of developers applying sub-optimal solutions [43]. This often happens when solutions that are easy to implement in the short term are used instead of taking the time to design the best overall solution. The term was introduced by Cunningham [13], who compared it to it taking a financial loan: interest accumulates over time, and at some point the debt will have to be paid or it will have negative impact on the pace of development. Technical debt can be seen as the collection of invisible, negative technicalities in the software, as illustrated in Figure 2.1.

Technical debt accumulates when the development team takes shortcuts, makes quick fixes and skips writing tests, causing the code to become harder and harder to maintain [12]. For example, this may happen due to business pressures, lack of understanding, poor collaboration or delayed refactoring. Technical debt accumulates over time and causes the development pace to slow down. Eventually it may make the developers afraid of introducing changes to hard-to-understand parts of the codebase in fear of breaking functionality. According to Crispin and Gregory [12], this fear may exist because

of not understanding the underlying code or because of the lack of tests to catch mistakes.

Analogous to technical debt, there can also be *social debt*, defined by Tamburri et al. [58] as "the unforeseen project cost connected to sub-optimal organizational-social structures". These can appear as issues like lack of communication, unresponsive or egocentric team members, or problems related to cultural differences. Tamburri and Di Nitto [56] argue software architecture to have a key role in the formation of social debt, and that studying social debt in architecture level is as important as studying technical debt to reduce waste during development. They continue to define *architecture incommunicability* as the inability to communicate architecture decisions to those who should be aware of them.

These two should not be seen in isolation as they have strong correlation [43, 56]. Cultural problems easily lead to technical debt, and vice versa.

	Visible	Invisible
Positive value	Feature	Architecture
Negative value	Defect	Technical debt

Figure 2.1: Technical debt can be seen as the invisible technicalities that provide negative value. [57]

2.1.2 Design smells and software maintainability

Two factors have been argued to affect maintainability: maintenance tasks to be performed, and the people who are to perform those tasks [4]. In the software industry it is relatively common for people to switch from different jobs and projects frequently, and developers often get assigned to existing

projects. Therefore maintainability is essential for new project members to reach good performance in their work. As stated by Kleppman [33]:

> *Maintainability has many facets, but in essence it's about making life better for the engineering and operations teams who need to work with the system.*

Agile methodologies try to embrace changing requirements. Tools and patterns such as test-driven development and continuous integration have been developed to make maintenance easier. Kleppman [33] defines three design principles to help with maintenance:

- Operability — The system should be easy to keep operating smoothly, its health should be visible to the people maintaining it
- Simplicity — The system should be easy for new engineers to understand, complexity should be minimized
- Evolvability — It should be easy for new engineers to implement changes as the requirements change

Unmanageable, *rotten software* often ends up being targeted for a redesign, but such redesigns rarely succeed [39]. This phenomenon can be called *shooting a moving target*: the old system continues to evolve while the new design must keep up. This problem will likely accumulate before the new design even makes it to its first release. This was also a significant risk in the case project. Martin [39] defines seven design smells that define a rotting software:

1. Rigidity — The system is hard to change because every change forces changes elsewhere in the system
2. Fragility — Changes cause the system to break in places that have no conceptual relationship to the part that was changed
3. Immobility — Hard to separate the system into components that can be reused elsewhere
4. Viscosity — Doing things right is harder than doing things wrong
5. Needless Complexity — The design contains infrastructure that offers no direct benefit

6. Needless Repetition — The design contains repeating structures that could be unified under a single abstraction

7. Opacity — The code is hard to read and understand and does not express its intent well

Design smells all affect maintainability in a negative way. Many of the points listed here can be seen as causes of technical debt. According to Martin [39], the best way to fight this rot is to follow strict principles for code quality and not let the rot begin in the first place: keeping the codebase clean as part of everyday development, not by rushing features and doing cleanups later. This requires commitment and discipline from the developers.

People often dislike maintaining legacy systems due to their complexity. Complex software makes maintenance hard; when the system is hard for developers to understand, defects are introduced more often. For example, developed features may have unintended consequences somewhere else in the system. Complicated and changing dependencies are a common problem for many developers [43]. Reducing the amount of dependencies, while also keeping them up to date, can be seen as an important factor in ensuring the maintainability of the software as a whole.

Moseley and Marks [45] define complexity to be accidental if it is not inherent in the problem the software solves but arises from the implementation. Making the software easy to maintain does not need to be done in the expense of reduced functionality. However, Fenton and Pfleeger [19] suggest that technical complexity might have a positive impact on productivity in case of new development and a negative one in case of maintenance.

Measuring maintainability can be challenging. Technical debt and even the aforementioned design smells can be seen as quite vague. To gather measurable data on maintainability there needs to be quantifiable metrics to track. For the software itself, aspects such as complexity and performance can be measured objectively. Cyclomatic complexity is a measure of the maximum number of linearly independent circuits in a program control graph, which has been widely used in research [27]. The purpose of the cyclomatic complexity graph is to identify software modules that will be difficult to test or maintain [40, p. 435]. Counting the lines of code (LOC) used for the different parts of the software can also be used as a rough measure of the size of the system, however it has been argued to be a poor measure of complexity [20].

2.2 Developer motivation and team dynamic in agile teams

The original, *waterfall*-style software project management was a process-oriented, slow procedure that proceeded in distinct steps that followed one another, from requirements specification to development and finally delivering the actual product. The often criticized weakness of this is that the requirements rarely stay constant and the product may not be fit for use once it finally finishes.

In contrast to this, the Agile Manifesto values *responding to change over following a plan*. Agile methodologies explicitly integrate social aspects into software development, and the focus on people has been an important factor in their success. They offer an alternative to traditional, waterfall-style processes, much like the Lean manufacturing that was introduced in industrial projects. [9, 62]

According to Conradi and Fuggetta [11], developers are motivated for change and there often exists a consensus on what are the most critical areas to address for maintenance in a given software product. Improvement and learning cannot be forced from the outside; rather they must become an integral part of the development process.

2.2.1 Developer experience

Analogous to to user experience of the product itself, *developer experience* refers to the emotions of the developers involved in a software project. Fagerholm and Münch [18] define developer experience as a means of capturing how developers think and feel about their activities within their working environments. They argue that an improvement of the developer experience has a positive impact on sustained team and project performance.

Developer experience and social debt can be seen as similar concepts in many ways, both having a clear impact on the performance of the team. Many studies show that the human factors in software development are very important for the performance and quality of produced work. Abdel-Hamid [1] argues that flaws in communication and coordination will lead to a failed development effort.

Fagerholm and Münch [18] categorize developers' performance-affecting factors to three categories:

- task characteristics
- characteristics related to self-development

- material and safety factors

These include things such as technical challenge and problem solving, opportunity for personal growth, recognition, the importance of the work itself, responsibility, job security, benefits and salary — all of which are closely related to and affect motivation.

On a team level, important performance factors include a high level of technical competence, well documented work, sharing knowledge with the team, team synergy, and the ability to share a common vision [5, 62]. High-performing teams are proud of their achievements and technical competence. They also adapt well to the personalities of individual members, maintaining good communication. Trust and openness about problems are crucial in a high-performing team [5].

2.2.2 Kanban as a development workflow

The Kanban method has been an emerging trend in software development. Kanban is an approach to Lean software development. The Lean methodology, as well as Kanban, originate from the the Japanese car manufacturing industry in the 1950s. Due to their success they have been widely adopted into software development among other industries. [3]

One of the key characteristics of Lean production principles is minimizing all kinds of waste from the development process [31]. In this thinking, waste is everything that does not add customer value to the product. Table 2.1 highlights different kinds of waste in the context of software development, as defined by Ikonen et al. [31].

Kanban aims to increase productivity by reducing operational costs, shortening the feedback loop and time-to-market. The main idea of Kanban is to visualize the workflow in an efficient way, to limit work in progress (WIP) and to measure the time to complete items [31]. Visualization is done with a Kanban board by showing assigned work to each developer, clearly communicating priorities and highlighting bottlenecks [3]. Kanban itself is also a Japanese word for signboard.

The rise in popularity of the Kanban method stems from the highly positive results achieved in manufacturing industry. Adoption in the software industry has been wide and its characteristics of encouraging communication and cooperation within the team, as well as the ability to adapt to changing requirements are seen positively when compared to traditional project management methods.

In an ideal Kanban process, the WIP limits should be enforced and the amount of tasks in progress should stay constant to utilize available resources

without extra waiting. However, in practice this is hard to achieve. A visualization of such an ideal process is shown in Figure 2.2.

Kanban has been found to improve various aspects in software development, including the quality of produced software, customer satisfaction, time to fix defects, motivation, productivity and communication between stakeholders [3]. The simplicity of the Kanban process pushes teams to communicate and coordinate their work, which allows for a better understanding of the whole development process and where the development is heading. This has been shown to motivate team members and improve efficiency [3].

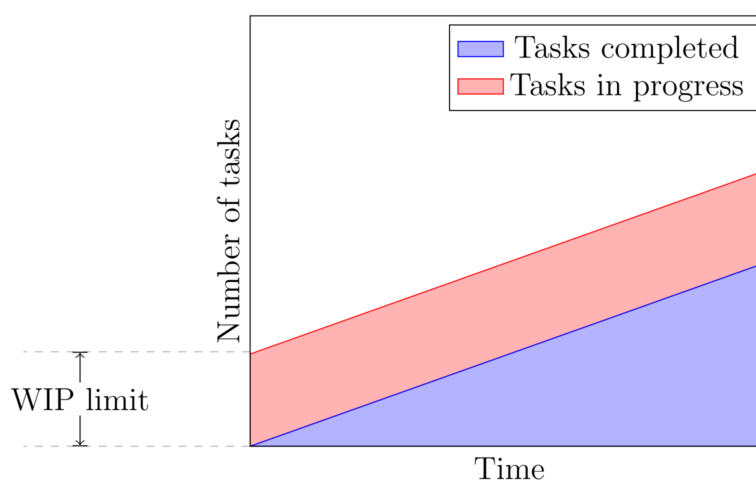

Figure 2.2: Ideal task progression in the Kanban process.

2.3 Risk management and the cost of change

The need to make small, incremental changes to mitigate the cost of mistakes has been widely recognized as a way to manage risks in software development [30, 48, 50]. People can *and will* make mistakes. Instead of delivering multiple major updates at once, incremental updates allow to better pinpoint problem areas. Newman [48] sees a correlation between a large cost of change and increased risk.

2.3.1 Continuous delivery and -integration

Software as a Service (SaaS) is a model of software delivery where the software is used over the Internet, without the end user having any physical access to

Element	Rationale for considering as waste
Partially done work	• Does it work and really solve the business problem? • Ties up resources
Extra processes	• Unnecessary paperwork consumes resources and adds no value for the customer
Extra features	• Tracking, compiling, integrating and testing an extra feature consumes resources • Potential failure point
Task switching	• Working in multiple teams causes more interruptions • Re-orientation back to work takes time
Waiting	• Delays in starting the project, staffing, reviews, approvals, testing, etc. add no value • Prevents realizing value for the customer as fast as possible
Motion	• Lack of immediate access to other developers and appropriate representatives disrupts concentration and re-establishing focus takes time • Tacit knowledge regarding artifacts (e.g. documents or code) does not move with the handoffs between people
Defects	• A minor defect discovered after weeks is more time-consuming than a major defect detected in a minute

Table 2.1: Sources of waste in software development [31]

the software executable itself. This makes the software product seem more like a service to the user while also making it easier to take into use. The SaaS-model became globally known in the mid 2000s [46]. Since then, the global SaaS market has grown to an estimated $58 billion in 2016, with an expected increase of 21 percent for 2017 [25].

In a SaaS-product, rolling out changes can be done in small increments, and new features easily tested with a limited set of users. This enables the provider to do rapid development and deliver constant updates to the product. Before the widespread adoption of the SaaS-model, updates were commonly delivered in bigger milestones. For example, Microsoft used so-called service packs to deliver accumulated fixes and updates to the Office suite. Delivering applications online allowed to tighten the release cycle and move towards continuous integration (CI) and -delivery.

Continuous integration has gained a major foothold in software development and become a mainstream technique. The term itself was originally introduced as a part of the Extreme Programming development process, being one of its twelve practices. Fowler [21] introduced the method with more detail in 2006. He defines it as a practice where team members integrate their work frequently, with each integration being verified by an automatic build process. The goal is to make sure that newly checked-in code integrates with existing code.

Newman [48, p. 103] mentions continuous integration as a key practice to make changes quickly and easily, with the goal of keeping everyone in the development team in sync. He also highlights the benefit it brings of receiving fast feedback on the quality of committed code. Crispin and Gregory [12] say that an automated build and integration process to run unit tests is a must to minimize technical debt. Keeping technical debt to a minimum, in part by applying agile principles, will free resources from the team to maintain a high quality product.

Similarly, Fowler [21] highlights reduced risk as the greatest benefit of continuous integration. It eliminates "blind spots" by having the development team always be aware of the state of the developed branch: what works, what does not and what defects are present. The defects are also easier and faster to fix, assuming the test suite is complete enough to catch them. This gives more confidence to deploy changes frequently, which allows to deliver new features and fixes more often. Thus, a comprehensive automated test suite can be seen as a prerequisite for continuous delivery [30].

Duvall et al. [17] state five key elements as the value of continuous integration:

1. Reduction of risks

2. Reduction of manual processes
3. Ability to deploy the software at any given time
4. Better project visibility
5. Greater confidence in the product

In addition to reducing risks and improving productivity by allowing for a more streamlined workflow, continuous integration eliminates uncertainty about the quality of the software and potential defects. Furthermore, continuous integration is not only limited to testing the software for defects. It can be used to maintain coding standards, such as analyzing the complexity, test coverage and performance of the software [17]. Duvall et al. [17] also point out performance as a key element of a good CI system. Since the goal of CI is to get timely feedback on the code quality, the faster the integration runs, the faster it makes the feedback cycle. A fast CI system has the potential to reduce unnecessary waiting in the project by a significant amount.

2.3.2 Microservice-architecture

Microservices are a recent trend in software development where the product is split into small pieces, *services*. Each service should be as small as possible with a clear responsibility. The concept has also been argued to be an implementation approach to Service-Oriented Architectures (SOA). In SOA, services are to be made inter-operable, hiding details of the execution environments behind them, aiming to have loose technical coupling between the services [47]. Microservices extend on the idea by applying modern software engineering paradigms such as RESTful HTTP, cloud computing and a continuous delivery approach to service delivery [48, 63].

This can be seen as an alternative to building single executable artifacts known as *monoliths*. Dragoni et al. [16] define monolith as "a software application composed of modules that are not independent from the application to which they belong". They continue to list common obstacles with monoliths:

1. Large monoliths are difficult to maintain due to their complexity
2. Dependency management is complex
3. Change in a single module requires restarting the whole application
4. Deployment and choosing the environment is often sub-optimal
5. They limit scalability

6. Technological lock-in to the language and frameworks of the original application

These can be seen as problems that begin to surface once the monolith grows large enough in size and complexity.

Newman [48] says microservices emerged as a trend from real world use. Many organizations have found that by embracing microservice architectures they can deliver software faster and embrace newer technologies. Newman [48] sees seven key principles for microservice-architectures as shown in Figure 2.3.

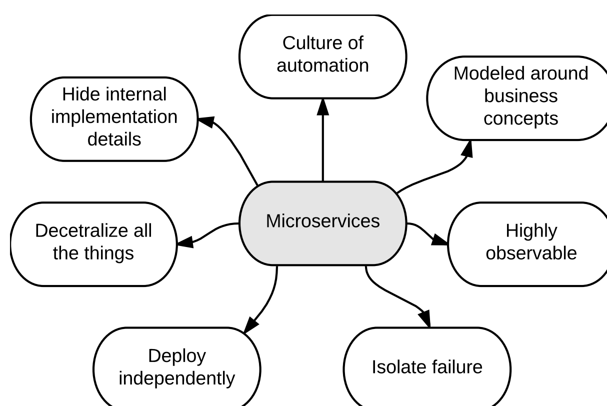

Figure 2.3: Principles of microservices [48]

The key takeaway from the principles is that the aim is to change the architecture towards a more failure-proof, decentralized and decoupled system that allows for independent deployments and embraces a culture of automation. Microservices are designed to withstand failure [63]; the loose coupling between services results in a more fault-tolerant system.

The flip side of this decoupling is that a microservice architecture is essentially a distributed system and the challenges and complexities of distributed systems apply. Even the proponents agree that they are hard to implement properly [49, 63]. Distributed systems also bring with them *partial failure*: some parts of the system may work fine while others are broken. The challenge with these failures is that they are nondeterministic; actions involving multiple services that sometimes work may sometimes fail.

Taking their drawbacks into account, it is clear that microservices are not a silver-bullet solution that should be applied to all projects. Table 2.2 compares maintenance-related aspects of a monolith and microservice-architecture.

Element	**Monolith**	**Microservices**
Deploying	Sub-optimal, change in a single module requires deploying and restarting the whole application. Some modules may be e.g. memory-bound while others are computationally intensive, yet resources need to be allocated for the whole monolith	Each service can be independently deployed and hosted in an environment applicable for said service
Scalability	Horizontal; new instances of the monolith required	Vertical; each service is responsible for one or more closely related functions
Maintainability	Technological lock-in to the language and frameworks of the original application	Each service can use the technologies suitable for the use case
Dependency management	Complex; updating libraries of a module may result in inconsistency in other modules	Each service handles its own dependencies independently

Table 2.2: Characteristics of microservices and monolithic systems

Chapter 3

The Kaiku Health rewrite project

This chapter gives background on Kaiku Health as a product and examines the rewrite project as a part of software maintenance: a major factor in taking on the rewrite was to also improve the codebase. We discuss the technical details involving the rewrite and the architectural decisions that were made, as well as the practical arrangements of the actual rollout. This chapter also introduces the development workflow used by the company during the the project.

3.1 Kaiku Health as a product

Kaiku Health's main focus is on gathering patient reported outcome measures (PROMs) and improving the relationships and communication between patients and their medical staff. Clinicians are typically not well trained in information science, and due to time constraints they are often resistant to take new systems to use and adjust their workflows. On the other hand, patients are not versed on medical terms; their symptoms are subjective and often described in various ways. Even imaging and dignostic tests often have room for interpretation, and diagnosis can be seen as a craft as well as science. [61]

PROMs originate from the need to involve patients with their own care: The goal is to gather patients' own view to assess the outcome of care they have received, analyzing the patient's health at different times in the care process. According to Black [8], use of PROMs has the potential to transform healthcare by helping patients and clinicians make better decisions.

Performance indicators are more unclear in healthcare than in industrial sectors, where metrics like revenue are universally agreed upon indicators of a company's success. In healthcare, the indicators are more subjective and

debatable; for example, comparing patients' quality of life or the quality of care is hard to do objectively.

Kaiku Health is delivered as a Software-as-a-Service (SaaS). Mäkilä et al. [46] define five characteristics that are typically associated with a SaaS-product:

1. Product is used through a web browser.
2. Product is not tailor made to each customer.
3. The product does not include software that needs to be installed at the customer's location.
4. The product does not require special integration and installation work.
5. The pricing of the product is based on actual usage of the software.

However, the study also states that businesses don't implement SaaS in a uniform way, and capturing a simple set of criteria is difficult. For Kaiku Health, not all of the criteria is fulfilled; since the product targets healthcare providers, often some customization and white-labeling is done, as well as integration to existing medical systems. For the end users, the product is used via a web browser. Pricing of the product is out of scope of this thesis.

3.2 Development workflow

In order to make conclusions about the success of the the rewrite project as a software engineering effort, it is essential to have some background on the development culture and workflow the case company had at the time. Being an early-stage startup company with a small development team, each developer had significant responsibility in a number of areas.

Development workflow was managed using the Kanban method. In Kaiku Health's case, Kanban was followed by having the product development visible on a whiteboard, with post-it notes describing an item that is in progress. Each employee had a limited set of magnets that signified involvement in an item; these also functioned as a tool to limit the work in progress per person. Items would move on the table from left to right as they progressed. The table was split onto a handful of *swimlanes* that could be used to separate items belonging to different features. The team would gather around the Kanban table each morning to a short daily meeting, usually taking 5-10 minutes, in which the progress of all incomplete tasks was described by the people involved in them.

Ikonen et al. [31] define three rules for Kanban:

1. Visualize the workflow
2. Limit work in progress (WIP) at each stage
3. Measure cycle time (i.e. the time to complete an item)

Comparing Kaiku Health's Kanban workflow to the aforementioned rules, points 1. and 2. are clearly followed. Measuring item cycle time was not done explicitly, however the Kanban table's visualization and the used development tools allowed it to some degree.

3.3 Release management

In Kaiku Health's case, new releases of the product were often deployed to production multiple times per week. This poses a significant risk of failure as well as a requirement for extensive quality assurance (QA). Service Level Agreements (SLAs) have to be obeyed not to cause inconvenience for the end users and possible financial and credibility losses for the company. In case a severe defect makes its way to production environment, patient information and care could potentially be at risk. This puts a lot of pressure on testing and quality control, which on the other hand forces developers to be responsible about their code and strive for quality.

The company used a variety of tools and methods to ensure that the product stayed reliable and no defects would make it to production. As the software used continuous delivery with all maintenance and upkeep handled by the delivering party, the traditional release cycle did not apply. Instead focus was on constant, high quality code review and deploying updates as frequently as possible.

Version control was handled using the *feature branch workflow*, which has been commonly used in in conjunction with Kanban [30]. In this workflow, each feature is developed in a separate branch and merged to mainline once complete. The aim of this workflow is to always keep mainline in a releasable state and to minimize interference between developers working on different features. Figure 3.1 shows the standard process Kaiku Health had for implementing a new code change into the product.

If a feature was deemed to be lacking in quality, the person responsible for reviewing it would leave notes for the developer to act on. This development-review cycle would continue for as long as the feature passed the level of quality required. However, in practice, no system is fool-proof and mistakes happen. The strictness of this process was often dependent on

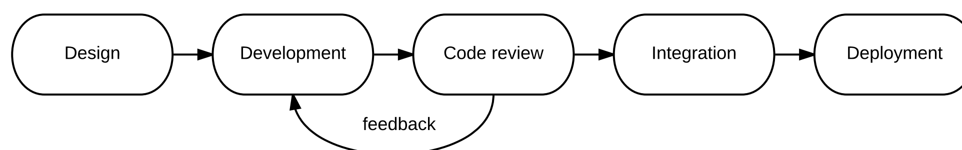

Figure 3.1: The feature implementation flow for the case company

the person reviewing, and other factors such as urgency for the feature to reach production sometimes could affect the level of scrutiny done.

3.4 The old architecture

Prior to the introduction of Single Page Applications (SPAs), web applications often suffered from poor interactivity and responsiveness towards end users [44]. Interaction was based on the user having to refresh the whole user interface in order to navigate or do actions within the application. This was also the case for Kaiku Health.

Prior to this rewrite project, Kaiku Health was a monolithic Ruby on Rails application with a minor amount of JavaScript used to add interactivity to the user interface. The old codebase used the standard Model-View-Controller (MVC) design pattern. In this pattern, which is very common especially for web applications, the software is partitioned into three distinct areas that each handle a distinct responsibility. The key idea is to separate user interfaces from the underlying data they represent [34]. Many popular web frameworks, such as Ruby on Rails and Django, encourage the use of the MVC model. Models serve as a representation of the data that the software holds and they handle the interaction with the underlying database. Controllers respond to requests by interacting with models and eventually responding with a view. This process is illustrated in Figure 3.2.

3.5 Objectives of the rewrite

Business-wise the main objective was to improve the user experience of the Kaiku Health application by updating to a new visual style and transitioning to a single-page application that would make user interactions smoother. The old user interface was deemed unncessarily complex, which was a pain point that provided the initial motivation for starting the rewrite. A major driver was also the emphasis on mobile use — the old interface had a subpar

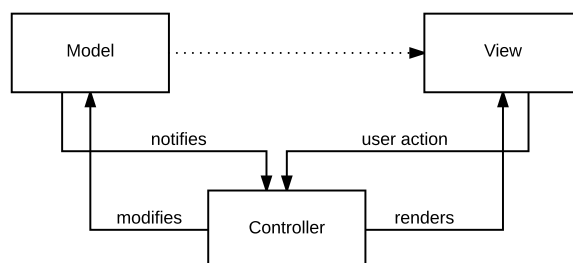

Figure 3.2: The Model-View-Controller design pattern.

experience with mobile devices, and a new design could help alleviate the problem while a native mobile application was still out of scope. As a major part of the user base consisted of elderly people, things like font sizes and contrasts needed careful attention to detail to ensure good usability. The aim was also to heavily simplify the application by reducing the amount of views visible to clients in order to make it easier to navigate.

Technically the main objective was to modernize the technical stack used for the application. The aim was to also take a leap towards a more reusable and maintainable microservice-architecture, as well as to improve on often overlooked metrics like developer happiness and productivity by transitioning away from a large part of legacy code.

3.6 Design phase and risk assessment

Architecturally the new frontend was chosen to be implemented as a single-page application. This would be developed using ECMAScript 2015, essentially a new version of the JavaScript language. The application itself would mainly depend on two libraries: React for the user interface and Redux to ensure unidirectional data flow. A move towards more client-oriented single-page application would be architecturally somewhat more complex, but were it to succeed, would provide notable benefits for the user experience and future-proofing of the product itself.

The architectural transition would be to move from server-rendered HTML views to having the user interface as its own, independent application that consumed data through an Application Programming Interface (API). In practice, to some degree this would mean separation of the frontend code from the Ruby on Rails stack. The target architecture is illustrated in Figure 3.3.

The main benefit of moving towards this model is to allow for a feature-rich frontend application while still minimizing its complexity by making state management simple. Functional programming has been argued to improve testability and reduce the complexity brought by state management [45]. By regarding all data as immutable, state management can be handled in a purely functional way. Same input always produces the same output, and the Redux *reducers* can be thought of as pure functions:

$$(state, action) \Rightarrow newState$$

This results in the application's state changes to always happen as a function of previous state and the action to be handled. For resolving defects, this has the added benefit that state changes can be travelled back and forth, allowing for the developers to easily see which part of the code is causing problems.

A major downside of a rewrite this major was that its effects would not immediately show: the way it was chosen to be done, most of the work had to be complete before it could be rolled out to end users. This was somewhat in contradiction to the principle of making small changes, advocated by Newman [48] among others. Due to major architectural changes it would not have been viable to do the rollout gradually. Also, the user interface was to be overhauled so that the old views would not be compatible with the new design.

The company took a significant risk in taking up on the rewrite in the way how it was done. The possibility of failure was acknowledged, as well as the effect it would have on getting deliverable results to customers. Developer resources would get constrained on the research and development of the new interface and away from building new features. Business-wise this of course meant that the perceived pace of product development during this transition period could seem slow for the end users: a significant portion of the limited development resources was to be constrained on this, which for a startup company meant that it had to be taken away from developing major new features. The pros and cons were discussed with the whole product team, and going for the full rewrite was chosen unanimously.

3.7 Development phase

The rewrite work was begun in May 2016 with a single developer. At the time, the development team consisted of a handful of people, most of whom were constrained on developing and maintaining different features. All progress in the project went through the company's standard peer review

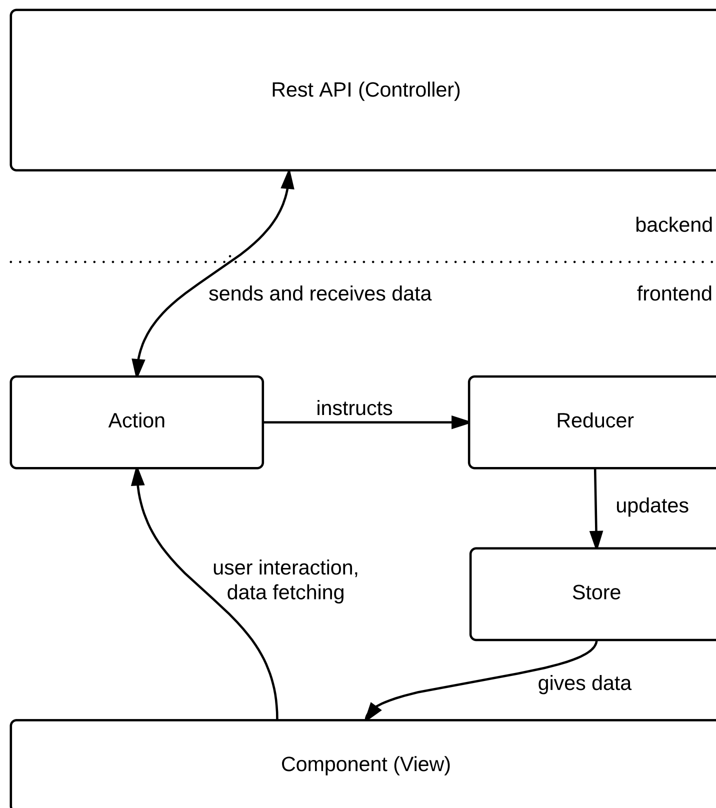

Figure 3.3: Data flow in the single-page application

process (Figure 3.1), and eventually more people got involved in the project as it progressed.

The development phase lasted close to a full year. No fixed delivery date was set in the beginning as there were a number of unknown variables that made predicting hard. The project was being built on technologies that were either not previously used by the company or had very minor usage, therefore some initial research had to be done prior to commencing with development. Also it was deemed more important to finish the project with satisfying results than meeting a certain deadline, even if it were to take longer.

3.8 Rollout

The original goal was to rollout the new interface to first customers in January of 2017, but this target was missed by a number of months. Finally, after

an internal testing and quality assurance (QA) period, the new frontend was gradually rolled out to customers during April-May 2017. The plan was to start with one customer that had a minimal amount of customized components, yet still a significant user base. As a positive surprise, no major defects were identified during the initial rollout. As the product was deemed stable enough, launch extended to majority of the customer base and to more specialized clinics that had a more comprehensive feature set in use. Overall, the rollout happened over the course of three weeks.

Chapter 4

Research strategy

This chapter introduces the methods used in the research and the objectives and scope of this case study. We cover the specifics of the preparation and collection of data that took place for this study.

4.1 Objective of the study

According to Lethbridge et al. [36], "to truly understand software engineering, it is imperative to study people — software practitioners as they solve real software engineering problems in real environments". Conducting studies is a practical way to do that: confronting the people that create and maintain software in their own environment.

This is a holistic case study of the rewrite project of Kaiku Health. The objective of the study is to gain insight into the details of how a major maintenance work affects the dynamics of a small development team, what should be taken into consideration when taking on such a task and what could be learned from choices and mistakes made. Focus is on the reflections of the development team. The key objective is to find out what are the main factors that affect the successfulness of a large product update such as this. Part of this case study is to investigate the culture at Kaiku Health, if technical or social debt was present, and to what degree.

4.2 Data collection techniques

This study is conducted as a *postmortem* of the case project in order to assess what issues were faced and what could have been improved in the development process. Postmortems are a way of looking into a project when it has

ended a phase or is terminated. They can be used as a collective learning activity to improve future behaviour — team members of a project always gain knowledge and experience that can benefit the individuals' personal growth as well as the organization as a whole [7]. According to Desouza et al. [15], project members must seek ways to improve on past experiences: "in order to prevent repeating mistakes, we must pay attention to the process of software projects". To learn as an organization, tacit learnings from individuals should be captured in an explicit format [15].

The study is based on the principles for software engineering case studies introduced by Runeson and Höst [52]. According to them, an inductive approach fits best for an investigative case study. They classify the gathered data to quantitative and qualitative. Quantitative data includes numeric, measurable information, that can be used e.g. to gather statistics. Qualitative data is less formal and can be textual, for example. This study focuses on gathering qualitative data.

According to Lethbridge et al. [36], data collection methods in a case study can be divided into three levels. First degree means that there is direct, real time contact between the researcher and the subjects. The second degree is for indirect collection without actual interaction with the subjects, i.e. via observing. The third degree is for independent analysis of produced artifacts and compiled data, such as requirements specifications that were not specifically produced for the study. Second degree contact differs from the third degree in that it requires data collection when work is occurring. The differences between the three techniques are illustrated in Figure 4.1.

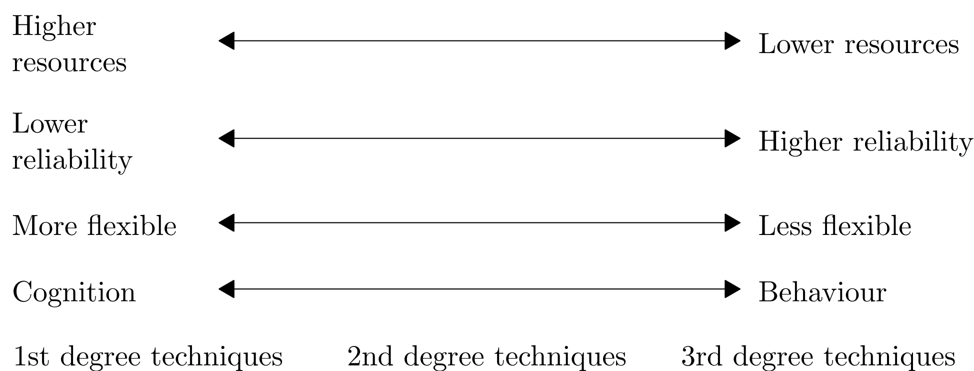

Figure 4.1: Data collection techniques compared [36]

Data collection in this study is done with direct one-on-one interviews, i.e. a first degree method. Lethbridge et al. [36] argue that first degree methods are invaluable due to their flexibility: software engineers can be

asked about a wide range of topics. Compared to just analyzing quantitative data, a much more complete insight can be gained. Interviews can be used to determine how enjoyable or motivating different tools, activities and ways of working are. There of course exists the downside that humans, by nature, tend to not be very reliable reporters and past events are often remembered with less accuracy.

Collected data is kept confidential and only scientifically significant background of the interviewees may be published. At the beginning of each interview, the interviewees were informed of their rights in the study and that the study is voluntary. All of the interviews except for one were conducted in Kaiku Health's office in Helsinki during October 2017, with the remaining one held over a Skype voice call, also in October. The interviews were held retrospectively after the project had completed in order to gain insight on the success of the project as a whole, as well as to assess how the interviewees perceived different aspects of the development process and what they thought had room for improvement.

Runeson and Höst [52] as well as Stake [53] emphasize the importance of triangulation in order to increase the precision of empirical research. That means taking different angles to gain a more comprehensive picture of the studied object. In this study, we use *data triangulation* that is done by having multiple interviewees whose responses are studied; this provides interesting insights from multiple viewpoints.

4.3 Interviewee selection

In order to ensure that the interviews provide meaningful insight, the selection of subjects is limited to people that in some way were involved in the project and were aware of its status during the development effort. All interviewees were working for the case company at the time. People with different roles in the team were chosen in order to gain multiple viewpoints into the project.

Five employees of the case company were interviewed about their perceptions about the rewrite project and about the company culture involving development in general. The interviewees included two software developers, one user experience designer and two people in management roles. The interviewees' involvement in the frontend rewrite project varied; not all were involved in the day-to-day operations. This provided an interesting look into how well communication worked between the parties and how they perceived the success in different ways.

Interviewing customers was scoped out of this study in order to focus

on the software development aspects. Due to the application being used for the end users' healthcare, the amount of usage varies a lot depending on the client; healthy users rarely have the need or interest to log in to the service unless they are asked to e.g. fill a questionnaire or are waiting for an answer. Therefore comparison between different clients might not provide comparable data, which is also a reason why they were scoped out of the interviewee pool.

4.4 Structure of the interviews

As Lethbridge et al. [36] state, the interview questions must be chosen carefully to ensure that the collected data is meaningful; poorly worded question may result in ambiguous answers that are hard to interpret. The interviews involved a series of open and closed questions, in which the interviewees were asked about their perception of the frontend rewrite project and ways of working during the project. This method of data gathering can be classified as a *semi-structured* interview [51].

As Runeson and Höst [52] warn, sensitive questions about interviewees' own competence and opinions about coworkers are handled with extra care and kept to a minimum to ensure that no bias comes from a lack of trust between the researcher and the subjects.

Focus is on studying the sources of waste in the project and comparing them to the ones identified by Ikonen et al. [31] as shown in Chapter 2. Part of the interview was formed according to this. The study also aims to find how the interviewees perceive the importance of maintaining good architecture and minimizing technical and social debt. Finally, the interviews were concluded by asking about how the interviewees felt about the overall successfulness of the project and what lessons they had learned especially in the context of a startup in general. Topics of the interviews included:

1. Risk management of the project
2. Perceived technical debt and the effect of the rewrite to it
3. Team dynamic and success of the project management
4. Architectural success and stability of the finished product
5. Opinion on moving towards a microservice architecture
6. Success of the project and its schedule
7. Success of the rollout

Section	Description
Introduction	Background of the interviewee, history and role at the company
Planning phase	How did the interviewee perceive the planning and risk management of the project as well as the significance of the project
Development phase	How did the interviewee perceive the development phase: communication, methods of working, efficiency, waste, task switching. Questions about the development workflow for developers
Delivery phase	Did the rollout go according to plan, project successfulness in general, did it stay in schedule, did the product work as intended
Reflection	Key things learned from the project, ideas for improvement
Closing	Thanking the interviewee

Table 4.1: Interview structure

Questions about team dynamic and processes were added to try and find how the internal communication and performance of the team possibly affected project success.

The structure of the interviews is shown in Table 4.1, and full interview template (in Finnish) can be found in Appendix A.

Chapter 5

Empirical study: Kaiku Health user interface overhaul

This chapter looks into the results gathered from the conducted interviews. We try to find common themes the interviewees brought up and relate those to the literature studied in Chapter 2. The results are grouped into several themes that are relevant for the study.

5.1 Planning and risk analysis

The interviewees were asked about how they perceived the risk analysis done for the project, if they felt it was sufficient and if any identified risks were eventually realized. Most interviewees acknowledged that the formal risk analysis was quite light, but many risks were still identified:

1. Failure to deliver a technically functioning product
2. Failure to meet the business-wise required schedule
3. Leaving out essential features
4. Failure to improve the user experience
5. Risking patient safety

The identified risks were all of major significance; any of them being realized could be considered as a failure of the project as a whole. Failure to deliver would have made the project a lost effort on top of consuming a lot of resources: this could be considered the most major risk in terms of how hard it is to recover from. Failure to finish the project on time was seen as important business-wise, but it was not the highest priority during

development. No strict deadlines were set at any point; focus was instead on releasing a well-polished result. The project's success was seen to have major importance due to it having consumed a large amount of time and resources:

> The way I see it, the risks we had there, they did not realize [...] if something so urgent had come up that we would have had to leave the project incomplete, like dragging on... that might have been the greatest risk, considering how much time we had to allocate to it. (Participant E)

> Well the risks we had, you can't really get rid of them if you want to evolve the software, make it better [...] if we had made really extensive plans, it might still be under construction...
> (Participant A)

Leaving out essential features was seen as a risk due to the user interface itself seeing significant re-design and many views within the user interface being simplified:

> We made quite bold choices, like getting rid of majority of the patients' views [...] it was the kind of risk that we considered if we should roll it out really carefully and see what happens...
> (Participant D)

The scope of the project was considered really large in context to the resources available in the company at the time. Overall, the project took nearly a year to complete, during which updates to the old user interface were a low priority.

5.2 The amount of waste

The interviewees were asked about their opinion on the efficiency of work done for the project and how it could have been improved. Looking at Martin's signs of code smells (Table 2.1), several of them could be identified from the interviewees' answers. We categorize the answers to four categories: *partially done work and task switching*, *defects*, *extra processes* and *waiting*.

5.2.1 Partially done work and task switching

The interviewees pointed out that there was a constant high amount of unfinished work. When asked if it it was common to have unfinished tasks before a new task assignment, one interviewee responded:

> Yeah, like constantly, having many things to do at once, like at the moment I have 6 projects I have to work on... then some others that I should work on, but there's no time.
> (Partipant D)

This was echoed by all of the interviewees; some found it a more acute problem than others. This was explained by the lack of resources, but it was also appointed that there was a high amount of interruptions happening:

> I think we have had a bit too much of that, and a lot of it in the daily meetings, you could see it when we got these bugs to fix, they were kind of thrown around... (Participant E)

> Well of course there's these small tasks and interruptions, especially in a company of this size... there are things you have to do, then there come a bit more important things, then you do those with a bit of hurry...
> (Participant A)

5.2.2 Defects

Test coverage was seen as something that could have been improved, however the new codebase was shown to be quite reliable and not that prone to errors:

> I can't think of any major showstoppers, we tested it internally quite a lot before release... it might have been more efficient to test it more during development, if we had planned more and divided the implementation to smaller pieces [...] the manual, internal user acceptance testing could have been done better. But the automatic tests seem to have a decent coverage, nothing too bad has gone all the way to production so far.
> (Participant E)

> There could be a lot more tests for it... then again, thinking of the release, there was a surprisingly low amount of defects. And the critical ones, we fixed those really fast, I can't remember a case that there would have been a major bug that would have affected a large amount of users [...] we were quite careful with the rollout, the first step was using it internally, there we found some bugs, then extending to selected customers, and so on [...] there was no apocalypse-like chaos with everything breaking down, no panic...
> (Participant B)

Interviewees seemed to have a consensus that the product was released without defects that would have caused major issues, which can be seen as contributing to the success of the project.

5.2.3 Extra processes

Interviewees were asked how they felt about the development workflow, if there were any extra processes or if something essential was missing. All interviewees pointed out that the company had a quite minimal formal processes when it came to planning development:

> We have a quite light process, if you can call it a process, but it works well for a team of this size, we get things done fast but I'm not sure... Maybe there would have been more benefits if we had better documentation... It comes down to onboarding, it would have been smoother to get accustomed to the codebase if you didn't need to reverse things from uncommented source, how things work... (Participant A)

> Well we didn't plan it that much... we had a quite clear idea of what we want in the end, in that sense we planned it, we knew what the goals were [...] but how big of an effort it would be, how much work it would require, that wasn't really assessed... (Participant B)

It wasn't seen as a major pain point, rather something that would need to be looked at when the company was to grow in size. This also reflected the difficulties in communicating requirements and progress of the development.

One interviewee pointed out that the efficiency of working could have been improved at some phases:

> I had a feeling that the last couple of months, we polished some a bit inessential things quite inefficiently, we maybe could have rolled it out a bit earlier... (Participant A)

5.2.4 Waiting

There was seen to be some waiting due to delays in the design process and a lack of resources:

> The waiting was maybe related to the design phase there, if we had had more people, more resources, we could have started

> the development earlier and they would have gone more hand-in-hand. (Participant B)

> Maybe the time was lost in having to do some things a second time [...] if there was a need to wait for specifications before starting to code something that you think is best, or waiting for acceptance for some already done features. (Participant D)

These wait times were seen as causing an amount of extra work due to miscommunicated specifications and developers implementing some features prior to complete specifications being ready.

5.3 Ways of working

The interviewees were asked about how they felt about the efficiency of the company's ways of working, such as the feature implementation process (Figure 3.1), team dynamic and used practices like Kanban and the daily meetings.

5.3.1 Code review and continuous integration

Code review was seen as a crucial element in completing features and fixing defects reliably:

> At time is may feel like this slows things down, it feels like things just stand still in review, but I haven't heard of a single software company where it wouldn't have been a problem at some point... it is a natural problem and you have to strive to remember that the things in review are kind of like a Ferrari in a garage, you don't want to hold it there for too long, but you still need to do the review well and look into it. (Participant B)

Similar thoughts were brought up by majority of the interviewees — no one questioned the importance of proper quality control and making sure the product stays stable. The workflow in implementing features was seen to be mostly adequate:

> Well I think our basic workflow, it has been ironed out to work quite well... the pull request -practices and such, they work quite well in my opinion, everyone usually does things in a quite smart way, and things get reviewed quite fast on average... sometimes not-so-urgent stuff drags on if something more important comes along, but that must happen everywhere... (Participant C)

5.3.2 Team dynamic

It was apparent from the conducted interviews that the way the development was conducted would not scale well for a large organization. Ways of working were felt as somewhat disorganized and there were significant signs of miscommunication — not everyone was aware of the progress, and specifications for features were sometimes incomplete or missing. This was viewed as something that should have been improved. It wasn't seen as an error in management, rather a constraint coming from the amount of work constantly being at a high level. A more strict enforcement of WIP limits was pointed out by two interviewees as a solution. However, there were no signs of a lack of motivation despite the challenges:

> I feel developing the product is motivating... the process should have been better though [...] Otherwise, I feel the team worked quite well, but uh... communication could have been improved, communicating if you did some feature and asking for comments. (Participant D)

A common find was that there was some siloing that happened in the development team, even though the team was small. Siloing people to a specific area of the product was deemed problematic and even more so when it happens in a small development team:

> Well of course there was some siloing [...] it would have been nice to have some more rotation in the people working on it. (Participant A)

Parts of the development team were developing significantly different areas of the product, which made them not that familiar with the new development, apart from the possible code review they did. However, it was acknowledged that this was mostly caused by a lack of resources within the company, and involving the whole development team in the rewrite project at the same time would not have been viable. The fact that the development was somewhat siloed was seen to have its positive sides too by reducing the amount of handovers between developers, letting the designated developers concentrate on the project:

> I think it [the development effort] was fine, it depends on the significance business-wise, how important it is to get out fast. We got it out in a decent time... If we had added more people to it, the value for time spent would have probably decreased, with too many cooks in the kitchen... (Participant A)

5.3.3 Tracking progress

When asked about how well the interviewees stayed on track of progress, none were concerned about it. As a generalization, the less involved in development the person was, the less they were aware of the details. However, this was not really seen as a problem:

> I think it was a good choice to keep the development team quite fixed [...] at some point, not everyone was that well aware of the progress because there kind of was the people dedicated to working on it... not many people [...] but I'm not sure if it was really needed at that point, for me getting familiar with the new code happened by inspecting it myself... (Participant B)

Then again, for the developers actively working on the rewrite, it was not always clear what work was to come:

> Well, I wasn't really concerned about that [progress], so I guess it was OK, maybe what was unclear was, it was clear what had been finished but it wasn't always obvious what was to come before it was ready to launch. (Participant A)

This clearly signifies that communication could have been improved. The company's ways of tracking progress, mainly the Kanban process and daily meetings were also brought up. All interviewees found them as a good way of keeping track for the most part, but some issues were brought up:

> Our daily is maybe a bit too fast-paced in a lot of things, you miss a lot of small things... (Participant D)

> Maybe with the Kanban, it shows that we have too many focus areas, it can be like... in the dailies, there is a lot of stuff that doesn't really involve you, then you're like... I know this task is progressing, but it's of not much use to me, that's why it might be better if we had less of these areas we focus on... or a bigger team. (Participant C)

> It's starting to reach its limits, the team has grown to a size that the sessions are starting to take time, going through the things in a quite shallow way, like here's a note with a cryptic message that doesn't really explain what it's about... like the point of the session is for everybody to know what's happening, the relevant parts. (Participant D)

It was pointed out that there was room for improvement in the way the daily meetings were conducted. Team members were often hurrying to update the progress of their tasks at the last minute, before or during the daily meeting, or just forgetting to update them. Also, often the actual meaning and effect of some tasks was not clear, particularly ones the said member was not involved in.

The fact that Kanban was followed with a physical board instead of e.g. an online tool was seen as a positive for the fact that it actually enforced the majority of the team to gather in the same location to catch up on progress in an open way. This has also been found as a recommended practice by prior research [3].

5.4 Success of the project

Evaluating the success for the rewritten user interface itself relied mainly on the amount of bugs detected and feedback from users. The company had bug reporting systems in place, and majority of defects were caught automatically or during a QA testing period prior to launch. The user interface itself received very minor amount of complaints from end users.

5.4.1 Technical success

As for the technical impact of the rewrite, the consensus was that it was an attempt to reduce technical debt. The opinions on how this succeeded varied between interviewees — it was agreed that it was pushing the product to a better direction, but the transition phase had its quirks:

> It has gotten more complex, but I think in the long run [...] we have taken some, like technical debt, we may have to rethink some things and how they fit the big picture in the long run, but I wouldn't say... like that we got it out in a reasonable time, you just can't think of all the things at once, and not all things go right at the first time. (Participant B)

Stopford et al. [54] highlight the importance that the development team understand the adverse consequences that come with technical debt: this requires awareness of the potential risks it can cause. When asked about the state of technical debt in the product, all interviewees acknowledged a level of technical debt, but none found it alarming. The overall complexity of the product was also pointed out:

> Well the new codebase is the only one that's really familiar to me [...] I find the Rails monolith kind of OK, but there are a lot of bells and whistles, things that no one has touched in ages or even seen them, and who knows what... (Participant A)

> Making changes to some places is really easy and to someplace else it can be really hard... there are some really complex parts. What's good is that the backend test coverage is high and that the tests will catch a lot of mistakes [...] but at some point we need to start hammering the complexity out. We still get changes made but it shows that some parts require a lot of domain-knowledge about healthcare and that makes it hard [...] I don't think there's a fast solution to that, and there doesn't need to be, we just need to make sure it evolves in the right direction. (Participant B)

The technical choices themselves were seen as successful, with no complaints about the architectural choices made. Mainly what was raising concern was the deployment and build process that had gotten more complex when splitting the frontend codebase from the Rails monolith:

> The biggest thing that surprised me was how much it affects the tooling, and somewhat the technologies that we already use [...] with these single page applications, there are not so clear, established ways of doing things, there are some many choices... (Participant B)

5.4.2 Transitioning to microservices

Microservices were seen as an interesting concept but not relevant to pursue at the time. This was explained by the size of the development team; the interviewees felt they would more likely bring unnecessary complexity rather than help with decoupling:

> I'm not too convinced about the microservice-thing, I see it as a good thing in a large organization [...] but if it's a small project and organization, many have just shot themselves in the foot with it [...] you solve some problems but create new ones also. (Participant A)

> It's not clear to me what problems it would solve, or those are a bit hypothetical at this point [...] Microservices in some things... we have such a small company, not so many people, the benefit of them couldn't be realized fully... (Participant B)

The mentality was clearly a bit hesitant overall, and the architectural change from a monolith was seen as a risky challenge. However, microservices were seen to have their positive sides too:

> When the frontend and backend are more separated, it's really good, you can think of the backend as just the API...
> (Participant C)

5.4.3 Essential factors in a successful launch

The interviewees were asked what they thought was essential to successfully finishing a similar software project and if they felt the case project was completed successfully. What came up was mostly related to project management and development practices:

1. A clear development process that is followed
2. A clear scope so the project stays on schedule
3. Communication between stakeholders and the development team
4. Visibility of progress

Points 1., 3. and 4. all revolve around good development practices and the importance of communication. The project scope was seen as something that should be clearly defined to reduce uncertainty and improve the level at which stakeholders stay aware of progress. Determining the right time to rollout the product was seen to have significant importance:

> It's easier for it to be late than for it to be released too early... the risk of it being late is much bigger. Of course if you release it when it's not ready at all and it just doesn't work, that is a risk too and there are examples of that all over the world, from software projects that released something that didn't work at all like they were expected to. But I don't know if it is a problem of the rollout itself then, then something else must have gone wrong. (Participant B)

> We spent a lot of time to test it internally, collected feedback from the field, clients and users, well... that we did the uh, gradual rollout, not just turning it on for all customers at once, and we informed about it, like the users and clients well before we did it.
> (Participant C)

Good planning and explicitly defined goals were seen to have high importance as well:

> What's essential would be, when you start doing a thing like this, to think of a vision, what are the goals you want to achieve, now it was that we want to make this look nicer, more simple and so on, but they weren't really explicitly said out loud. But it would be really good, like to talk about that, what are the goals we could have, for example it could be to reduce technical debt also. Kind of to think of the goals and think of the ways [...] then the way it would have been technically done, would have been different [...] and this is not limited to like the architecture- or tech stack, or well that too, but it's a much wider topic. (Participant D)

The fact that the rollout went according to plans with no major issues was seen as a achievement that is not so common in software projects. The importance of focus was also echoed by another interviewee:

> Clearly a thing to improve would be to have better focus, visibility into the development [...] doing things in a smart way, in smaller pieces, not too big ones as those tend to not stay on schedule... and to really think what features are essential for the product. (Participant B)

5.5 Summary of the interviews

Ahmad et al. [3] found collaboration and communication issues and the difficulty of managing WIP to be common challenges when adopting the Kanban process, and these can also be identified from the case project. The sentiment was similar between the interviewees. To summarize, all of the interviewees felt the project had been finished with satisfying results, but the processes of getting there had room for improvement. What raised concern was:

- Shallow risk analysis
- Visibility into the progress
- Coordination between design and development
- Limiting work in progress (WIP)
- Clear project scope

There was no significant difference in how people in different roles perceived the project as a whole, however they emphasized different things. The amount of work in progress was seen as a major inconvenience, resulting in a large amount of interruptions, which in turn had a negative effect on productivity. This affected the development of some milestones being done without proper plans. Visibility of progress was seen to be mostly sufficient, but for some people, especially in the management roles, it was at times weak.

The amount of work was found to surpass expectations:

> Like it always goes, what surprised me was that there was a lot more work than we thought and like usual, the last 10 % of the project took 90 % of the time. (Participant D)

This materialized in the project missing its plans for delivery by a number of months. Some of this could be attributed to defects and missing features detected in the final QA phase prior to release, however some of the the sub-optimal development practices most likely contributed to this also.

However, the consensus among interviewees was that that project was completed successfully regardless of the challenges that it faced. Looking into the interviewees' perception of success, technical aspects of the finished product could be identified as the most major component — a system that works according to its specification with a low amount of defects.

Chapter 6

Discussion

In this chapter, we discuss the insights gained from the conducted empirical research and answer the research questions based on the case study and prior research on the subject.

6.1 Risk assessment in a near-complete frontend rewrite

The first research question is:

> *RQ1: What risks are involved in a large rewrite project for a startup company?*

In retrospect, all interviewees agreed that there was a lot of room for improvement in the project planning phase of the case project. The risk assessment made was seen as quite shallow and the project was begun with mostly just a broad vision of what the end result should be. This can be seen as putting a lot of responsibility for the development team. I argue it has a positive effect, to some degree, on performance and creativity as the developers are free to experiment and realize their visions of what the project should achieve. However, this has the downside that it easily results in a considerable amount of waste in the process, increasing the probability of unnecessary work and waiting. It also heavily relies on the prerequisites that the development team communicates openly and that the team members are skilled and motivated.

Companies of different sizes need different approaches to project management depending on the growth stage they are in [6]. As a company grows, it creates friction for both people and processes. A key takeaway from the

conducted interviews was that the case company was still finding its ways of development and process management — therefore generalizing the findings from this study might prove to be inaccurate for a lot of cases. However, a certain level of chaos and disorganized development is often common for startups; after all, they are pushing to fit the market with limited resources and financial instability. This turbulent environment requires swiftness, effectiveness and ability to react to changes in the market [42]. This has the downside that in the case of startups, strategic plans tend to be ignored in favor of tactical wins [6].

The development process in the case project was not very thoroughly thought out at first, and it carried on for relatively long. The major downside of this was that a large part of the team was not deeply invested in the process, which might have delayed it from the optimistic expectations had at the beginning of the project. In the context of a software startup company with limited developer resources, this posed a significant risk of failing to deliver. On the other hand, not rushing into a major change of this scale also allowed to have time to test the software thoroughly, and in the end deliver satisfying results. However, it could be argued that in this context, a more strict process for development would improve the performance and motivation of the team.

Duvall et al. [17] state that many risks related to the quality of the software and project management can be mitigated with a comprehensive continuous integration setup, hence making CI an essential part of good risk management. These risks include late discovery of defects, lack of visibility and poor quality of the software itself. Reducing them brings confidence that the product can be deployed at all times and that the code quality stays high. The results of this study support this statement — continuous integration was found to be an essential part of the development process and maintaining confidence in the quality of the software.

6.1.1 The importance of quantifiable goals

Software engineering often lacks disciplines of effective measurement that are common to other fields of engineering. The lack of metrics is a common problem in software projects, and many authors have suggested that metrics should be a part of most software development efforts [19, 40, 50]. However, before commencing with gathering data, the company should know what they want to achieve: their goals [50].

When measurements are made, they are often done infrequently, inconsistently or incompletely [19]. For example, a project may have the goal of improving user experience or reliability without clearly defining what is re-

quired for these objectives. This has been referred to as Gilb's Principle of Fuzzy Targets [26]:

Projects without clear goals will not achieve their goals clearly.

Even though the context of this study was a large-scale maintenance project, a lot of the insights gained from this project are not specific to its technical aspects and could be applied to other resource-intensive software projects. I argue that in order to achieve *measurable* success, there needs to be a set of *quantifiable goals*. This is something that the case project can be seen as lacking — the targets were broad and not explicitly defined. With quantifiable goals, there should also be ways of measuring them. For example, performance can be measured by responsiveness of the system, usability by the amount of help the end users need and code by its complexity, test coverage and error rates.

Measurable targets such as these may seem unnecessary at first, but tracking the state of the product's attributes becomes more important as the product matures and reaches wider use. However, in many cases the collection of metrics data ends up in a massive data collection effort with very little analysis or reporting done [50]. The collection of metrics should always support the organizational goals and the gathered data should be analyzed in light of them.

Banfield et al. [6] suggest that having a lot of activity without clear metrics for measuring anything of value may result in so-called "scrum theater" — an illusion of productivity without actual material results. However, a high level of measurement can be seen as invasive by the team if it tries to measure individual developer effort [14], and this should not be the goal. Instead, measurements should be applied to detect changes in attributes such as the reliability, maintainability and performance of the product.

6.1.2 Architectural considerations and the suitability of microservices for a startup

The second research question is:

RQ2: How suitable are microservices for a startup company?

Naturally, the architectural choices made for a software project contribute to its riskiness and are of significant importance in terms of its success. A wrong technical choice may turn out to decrease the pace of development, deter some team members from getting invested in the new codebase or

introduce new technical problems later on. Combined, these are likely to result in an increasing amount of technical and social debt.

According to Glass [28], adapting to new technology has an initial learning curve that will result in a loss of productivity. This is followed by a slow improvement, and eventually the investment should pay itself off. Developers should be aware of the learning curve and be willing to tolerate it; expecting immediate benefits is often unrealistic.

A major obstacle with the microservice-approach is the fact that a system composed of microservices is essentially a distributed system and challenges related to distributed systems apply to it. Microservices bring increased complexity with them. Consistency in versioning and error handling needs to be handled carefully, as well maintaining of data integrity. For example, in the case project, the frontend and backend software versions need to always stay in sync, meaning that no major API changes can be made without updating the frontend.

Lewis [49] sees continuous integration and a fully automated build pipeline as a prerequisite for a microservice architecture; building microservices should only be considered if these requirements are met. He brings an example of a small team, much like in the case project, where microservices bring a premium in the required setting up of the infrastructure. This is something that is likely to slow the team down in the short term, and can be seen as unnecessary if the company is still validating its business. Fowler [22] shared these views and argues that the premium is so high that microservices should not even be considered unless the system is too complex to manage as a monolith. Instead of separating to microservices, focus should first be in making the monolith more modular and easier to manage.

Amundsen [49] says microservices should be deployed to enable business goals, meaning they should provide a clear business value. Lewis [49] argues that microservices themselves are just an optimization, and will not necessarily make the development-release cycle faster, where a good management of work in progress gets a long way:

> "How work flows from concept into production that is the first-order factor in getting software into the market." [49]

If there is a lot of waiting due to process constraints, like quality assurance and testing, the cycle will not be helped by changes in the infrastructure.

These thoughts were echoed by the interviewees of this study; the overhead of setting up a new infrastructure was seen as a too big of a task to take on at the time. This stance was mainly rationalized by the limited resources and questionable benefit the transition would provide. Also, the continuous

integration and deployment infrastructure was not seen as ready for it. However, it was seen as a good direction to explore and transition to in the long run.

Therefore, I find the suitability of a microservice-approach to depend a lot on the overhead it would bring to development. In the context of startups, the amount of this overhead can often be too much to try to justify. Especially if the software has already been developed as a monolith and no significant technical challenges are present due to this, microservices could be argued to be a premature optimization.

6.2 How to succeed in a major development effort within a startup

The third research question is:

> *RQ3: How to succeed in a large rewrite project and are they an efficient way to reduce technical debt?*

In general, success is relatively rare in software projects; Agarwal and Rathod [2] find differing perceptions as one of the reasons. Success means different things to different stakeholders. Defining success and failure in the context of software projects can be problematic, and in general there lacks consensus on how to define them. They are vague terms that are difficult to measure. However, a common measure of success of a software project is assessing its ability to meet the target cost, schedule and desired level of quality [2, 24].

Ideally projects will have successful processes and outcomes, but in practice this is often not the case. Markus and Mao [38] go as far as to separate *development success* from *implementation success*, and suggest that there is not necessarily a relationship between them. Interviews conducted by Linberg [37] suggested that even a project that failed to meet its implementational goals could be considered successful if it facilitated learning that could be carried to future projects.

Lehtinen et al. [35] researched common causes for software project failures and how they interconnected. They classified causes of failure to four areas: *tasks, methods, environment and people.* Each of these corresponded to an equal distribution in the amount of failures. Out of these, environmental causes were rarely seen as having feasible potential for process improvement.

Similarly, McLeod and MacDonell [41] introduced a framework to assess the success of software projects. They divided influencing factors into four dimensions:

1. Development processes; the activities associated with development, from requirements definition to management and implementation

2. People; both individuals and groups who are involved in the development project, that affect the decisions made through their actions and relationships

3. Project content; technological and strategic properties of the project itself, its scope and goals and resources allocated to it

4. Institutional context; factors related to the organization and the environment it operates in

They found that project outcomes often involve many of these factors and that in practice, all four dimensions are related and interactive. For example, the success of development practices often relies on the technical expertise of the people involved. We will inspect three of these aspects on software project success, leaving out the environmental factors due to the possibility of influencing those being minimal.

6.2.1 Development processes

Cockburn and Highsmith [10] argue that the turbulent nature of software development requires an agile process with responsive people and rigidity of the organization makes that difficult. Individual competence is of high importance, and “if the people are good enough, they can use almost any process and accomplish their assignment” [10]. Processes can provide a framework for teams to work together, but they cannot overcome a lack of competency, while skilled people can find their ways of working even with a inefficient process [6, 10]. Furthermore, individual developer productivity is found to decrease as the amount of people in the project increases [19]. The added people may even contribute to a decrease in quality of produced code.

In this study, the secondary importance of processes when compared to people could be identified. To some extent, the processes used could have been improved, but the development effort succeeded and reached its goals satisfyingly nevertheless. During development of the case project, many things were done right and the project progressed according to expectations regardless of some sub-optimal development practices such as the lack of a systematic design process. However, the somewhat chaotic process of development was seen to have multiple sides to it; visibility into the progress was at times quite weak and a lack of resources was apparent.

Ahmad et al. [3] suggested that creating a culture of collaboration on solving tasks and encouraging team members to provide feedback are key elements in overcoming issues in a Kanban process, as is providing the development team with a clear vision. In the case project, the visibility into the product roadmap was seen to be weak at times, which was pointed out to be problematic.

Banfield et al. [6] define six ways that a roadmap helps in delivering product work:

1. Focus; it helps understand what to give attention to
2. Alignment; it gets the entire team working toward the same goals
3. Prioritization; knowing the importance of each feature
4. Visibility; seeing the way the team works and what they will be doing
5. Coordination; minimizing overlapping efforts
6. Vision; how the product should deliver value to its users

A roadmap is not a replacement for a rigorous process or a good team, neither should it be a release plan with specific deadlines. Misaligned work reduces productivity, causing stress and waste [6], and that is a problem that a good roadmap can alleviate. *Opportunity for achievement* has been found to have an important effect for developer motivation [5], and the lack of visibility can make the development work seem as a collection of tasks rather than a project with a clear set of goals to achieve. Similarly, without a vision, a project can become a collection of solutions lacking a core problem [6]. Banfield et al. [6] suggest all features should be tied to strategic goals and thought of as a part of the product and not in a vacuum.

In light of the prior research and findings from this study, I argue that visibility into the project roadmap is essential for developer motivation and productivity, and eventually to the success of a software project as a whole.

6.2.2 People

The structure of software teams can reveal important information about what makes projects successful. Whitworth and Biddle [62] studied how the performance of agile teams is affected by the cohesiveness of the teams: awareness and commitment to common goals, importance of communication and sharing knowledge were all found to have an important role. Their research showed daily meetings to be an important motivator, increasing engagement

in the team. Open communication about progress and arising issues was seen to support feelings of satisfaction, acceptance and belonging. In teams where activities and issues were not regularly shared, failure to complete a task was instead associated with stress and annoyance. Some teams may stay together from project to project and build a team spirit that results in high productivity and motivation, while others fail to function well due to a lack of enthusiasm and communication.

Similarly, Fagerholm and Münch [18] found that success in software projects has a strong reliance on the people involved, while tools and methods only amplify the productivity of skilled and well-coordinated development teams. This is echoed by Agarwal and Rathod [2], who see managerial and organizational implications, personal growth and technical innovativeness more as variables contributing to success rather than measures of success itself.

Lack of motivation has been cited as one of the most common reasons for project failure [14, 23]. Software developers' motivation has a strong correlation to how they perceive the importance of their work and its quality [14, 23]. Thinking of software projects as a whole, motivated and competent team members can be considered as a criteria for success.

6.2.3 Project content

A software project needs to have realistic and achievable goals to have a chance of success in the first place. Wallace and Keil [60] argue that projects emphasizing outcome targets such as schedule will be managed differently to those that focus on product-related goals. No fixed delivery date was set for the case project, instead the goal was to finish the project as efficiently as possible while maintaining a high quality of produced work. A delivery target, or a prediction in general, is useful only if it can be made reasonably accurate.

Fenton and Pfleeger [19] reported on a study that found productivity to be the best for projects with no formally set completion target. The scheduling of the case project was not seen as problematic even though the project saw a delay in the quality assurance phase. Some delay could be attributed to specifications changing on the fly and there being an amount of waiting and extra work due to missing or incomplete specifications.

DeMarco and Lister [14] argued that too strict deadlines have negative effect on product quality:

> People under time pressure don't work better — they just work faster.

Developers will see this pressure as stressful and demotivating, which will ultimately affect the team performance in a negative way. Problems will be ignored to be dealt with later and they may even be knowingly let into the product. Conversely, quality is a means to higher productivity [14].

This idea is also shared by Banfield et al. [6], who emphasize the importance of team members being able to relate to another. People in such a team learn each other's strengths and weaknesses, helping them solve problems as a group. They claim this to result in the teams to be highly productive without any artificial pace: "The team's pace expresses their love for the craft of developing an amazing experience."

I suggest that setting an explicit delivery deadline should not be necessary when not necessitated by business agreements. The motivation to finish a project in time should come from internal attributes such as highly skilled and motivated developers, open communication and efficient work management that, when combined, reduce the amount of waste to a minimum.

Furthermore, I find the results of this study to mostly support earlier research about software projects and their success: success is hard to quantify, however a major part of success lies in skilled and motivated team with clearly defined goals and open communication.

6.3 Methodological considerations

I found the topic of the study an interesting and relevant area to research; optimizing the usage of these limited resources can determine the success or failure of an early stage company as a whole. The way this study was conducted proved to be quite challenging due to the small size of the company and the development team, which resulted in a somewhat narrow insight into the subject. The amount of interviewees was also low due to this. For a more in-depth look, it would be interesting to study these experiences from multiple companies of similar maturity, which would reduce the bias coming from individual project decisions and ways of working within each case company.

Chapter 7

Conclusions

The objective of this thesis was to find out how project and risk management could be improved in the context of an early stage software company going through a significant maintenance project. The literature study provided a look into common pitfalls in software maintenance and project management. Further insight was gathered by conducting postmortem interviews with members of the case project about their perception of its success.

The startup context often implies lower resources and less fixed ways of working compared to established companies. These may manifest as a somewhat chaotic culture in terms of development practices, mostly due to a constant hurry and low resources.

The results of this study indicate that there are a number of key elements that affect the success of a software project apart from the project content itself. These include *skilled and motivated individuals*, *open communication*, *a positive environment* and *development processes*. These are all heavily interconnected and affect each other, hence an improvement in one area can be expected to radiate a positive effect to other areas too.

I argue that in order to determine the success of a software project, its goals and scope should clearly defined prior to beginning with development. Furthermore, I suggest that a fixed delivery date is often not necessary; the level of productivity should come from internal factors such as the motivation and skills of the project team, which are often more important than the used processes.

For future research, I propose studying these topics in a wider setting with companies of different maturity. This could be used to determine the significance of the amount of available development resources. Additionally, a systematic literature review into startup software teams' success would help in discovering how unique the found challenges are to early stage companies.

Bibliography

[1] Tarek K. Abdel-Hamid. The slippery path to productivity improvement. *IEEE Software*, 13(4):43–52, 1996.

[2] Nitin Agarwal and Urvashi Rathod. Defining 'success' for software projects: An exploratory revelation. *International journal of project management*, 24(4):358–370, 2006.

[3] Muhammad Ovais Ahmad, Jouni Markkula, and Markku Oivo. Kanban in software development: A systematic literature review. In *Proceedings of the 2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '13, pages 9–16, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5091-6. doi: 10.1109/SEAA.2013.28. URL http://dx.doi.org/10.1109/SEAA.2013.28.

[4] Bente Anda. Assessing software system maintainability using structural measures and expert assessments. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 204–213. IEEE, 2007.

[5] Nathan Baddoo, Tracy Hall, and Dorota Jagielska. Software developer motivation in a high maturity company: a case study. *Software Process: Improvement and Practice*, 11(3):219–228, 2006. ISSN 1099-1670. doi: 10.1002/spip.265. URL http://dx.doi.org/10.1002/spip.265.

[6] Richard Banfield, Martin Eriksson, and Nate Walkingshaw. *Product Leadership: How Top Product Managers Launch Awesome Products and Build Successful Teams.* O'Reilly Media, Inc., 2017. ISBN 978-1-491-96060-8.

[7] Andreas Birk, Torgeir Dingsoyr, and Tor Stalhane. Postmortem: Never leave a project without it. *IEEE software*, 19(3):43–45, 2002.

[8] Nick Black. Patient reported outcome measures could help transform healthcare. *BMJ (Clinical research ed)*, 346:f167, 2013.

[9] Alistair Cockburn. *Agile software development*, volume 177. Addison-Wesley Boston, 2002.

[10] Alistair Cockburn and Jim Highsmith. Agile software development, the people factor. *Computer*, 34(11):131–133, 2001.

[11] H. Conradi and Alfonso Fuggetta. Improving software process improvement. *IEEE software*, 19(4):92–99, 2002.

[12] Lisa Crispin and Janet Gregory. *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[13] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992. ISSN 1055-6400. doi: 10.1145/157710.157715. URL `http://doi.acm.org/10.1145/157710.157715`.

[14] Tom DeMarco and Tim Lister. *Peopleware: Productive Projects and Teams (3rd Edition)*. Addison-Wesley Professional, 3rd edition, 2013. ISBN 0321934113, 9780321934116.

[15] Kevin C. Desouza, Torgeir Dingsoyr, and Yukika Awazu. Experiences with conducting project postmortems: Reports vs. stories and practitioner perspective. In *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*, pages 233c–233c. IEEE, 2005.

[16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016.

[17] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007. ISBN 0321336380.

[18] Fabian Fagerholm and Jürgen Münch. Developer experience: Concept and definition. In *Software and System Process (ICSSP), 2012 International Conference on*, pages 73–77. IEEE, 2012.

[19] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998. ISBN 0534954251.

[20] Martin Fowler. Cannot measure productivity. webpage, August 29 2003. `https://martinfowler.com/bliki/CannotMeasureProductivity.html`. Accessed 7.11.2017.

[21] Martin Fowler. Continuous integration. webpage, May 1 2006. `https://www.martinfowler.com/articles/continuousIntegration.html`. Accessed 1.11.2017.

[22] Martin Fowler. Microservice premium. webpage, May 13 2015. `https://martinfowler.com/bliki/MicroservicePremium.html`. Accessed 31.10.2017.

[23] A. César C. França, Tatiana B. Gouveia, Pedro C. F. Santos, Celio A. Santana, and Fabio Q. B. da Silva. Motivation in software engineering: A systematic review update. In *Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 154–163, Durham, UK, 2011. IET, IET. ISBN 978-1-84919-509-6. doi: 10.1049/ic.2011.0019.

[24] Mark Freeman and Peter Beale. Measuring project success. *Project Management Journal*, 23(1):8–17, 1992.

[25] Gartner. Gartner Forecasts Worldwide Public Cloud Services Revenue to Reach $260 Billion in 2017. webpage, 2017. `https://www.gartner.com/newsroom/id/3815165`. Accessed 26.10.2017.

[26] Tom Gilb. *Principles of Software Engineering Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN 0-20-119246-2.

[27] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284–1288, 1991.

[28] Robert L. Glass. The realities of software technology payoffs. *Commun. ACM*, 42(2):74–79, February 1999. ISSN 0001-0782. doi: 10.1145/293411.293481. URL `http://doi.acm.org/10.1145/293411.293481`.

[29] Penny Grubb and Armstrong A. Takang. *Software maintenance: concepts and practice*. World Scientific, 2003.

[30] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321601912, 9780321601919.

[31] Marko Ikonen, Petri Kettunen, Nilay Oza, and Pekka Abrahamsson. Exploring the sources of waste in kanban software development projects. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pages 376–381. IEEE, 2010.

[32] ISO/IEC. Software engineering — software life cycle processes — maintenance. Standard 14764-2006, IEEE, 2006.

[33] Martin Kleppman. *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 1st edition, 2017. ISBN 978-1-449-37332-0.

[34] Avraham Leff and James T. Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[35] Timo O. A. Lehtinen, Mika V. Mäntylä, Jari Vanhanen, Juha Itkonen, and Casper Lassenius. Perceived causes of software project failures - an analysis of their relationships. *Inf. Softw. Technol.*, 56(6):623–643, June 2014. ISSN 0950-5849. doi: 10.1016/j.infsof.2014.01.015. URL `http://dx.doi.org/10.1016/j.infsof.2014.01.015`.

[36] Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical software engineering*, 10(3):311–341, 2005.

[37] Kurt R. Linberg. Software developer perceptions about software project failure: a case study. *Journal of Systems and Software*, 49(2):177–192, 1999.

[38] M. Lynne Markus and Ji-Ye Mao. Participation in development and implementation - updating an old, tired concept for today's is contexts. *Journal of the Association for Information Systems*, 5(11):14, 2004.

[39] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0135974445.

[40] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, July 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837. URL `http://dx.doi.org/10.1109/TSE.1976.233837`.

[41] Laurie McLeod and Stephen G. MacDonell. Factors that affect software systems development project outcomes: A survey of research. *ACM*

Comput. Surv., 43(4):24:1–24:56, October 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978803. URL `http://doi.acm.org/10.1145/1978802.1978803`.

[42] Werner Mellis. Software quality management in turbulent times–are there alternatives to process oriented software quality management? *Software Quality Journal*, 7(3):277–295, 1998.

[43] Tom Mens. An ecosystemic and socio-technical view on software maintenance and evolution. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 1–8. IEEE, 2016.

[44] Ali Mesbah and Arie Van Deursen. Migrating multi-page web applications to single-page ajax interfaces. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 181–190. IEEE, 2007.

[45] Ben Moseley and Peter Marks. Out of the tar pit. *Software Practice Advancement (SPA)*, 2006.

[46] Tuomas Mäkilä, Antero Järvi, Mikko Rönkkö, and Jussi Nissilä. How to define software-as-a-service — an empirical study of finnish saas providers. In *International Conference of Software Business*, pages 115–124. Springer, 2010.

[47] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, 2004. ISBN 0321180860.

[48] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015. ISBN 1491950358, 9781491950357.

[49] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98, 2017.

[50] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996. ISBN 0471170011, 9780471170013.

[51] C. Robson. *Real World Research - A Resource for Social Scientists and Practitioner-Researchers*. Blackwell Publishing, Malden, second edition, 2002.

[52] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[53] Robert E. Stake. *The art of case study research*. Sage, 1995.

[54] Ben Stopford, Ken Wallace, and John Allspaw. Technical debt: Challenges and perspectives. *IEEE Software*, 34(4):79–81, 2017.

[55] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497. IEEE Computer Society Press, 1976.

[56] Damian A. Tamburri and Elisabetta Di Nitto. When software architecture leads to social debt. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 61–64. IEEE, 2015. doi: 10.1109/WICSA.2015.16.

[57] Damian A. Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. What is social debt in software engineering? In *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, pages 93–96. IEEE, 2013.

[58] Damian A. Tamburri, Philippe Kruchten, Patricia Lago, and Hans Van Vliet. Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications*, 6(1):1–17, 2015.

[59] Iris Vessey and Ron Weber. Some factors affecting program repair maintenance: an empirical study. *Communications of the ACM*, 26(2):128–134, 1983.

[60] Linda Wallace and Mark Keil. Software project risks and their effect on outcomes. *Commun. ACM*, 47(4):68–73, April 2004. ISSN 0001-0782. doi: 10.1145/975817.975819. URL `http://doi.acm.org/10.1145/975817.975819`.

[61] Jens H. Weber-Jahnke, Morgan Price, and James Williams. Software engineering in health care: Is it really different? and how to gain impact. In *Proceedings of the 5th International Workshop on Software Engineering in Health Care*, pages 1–4. IEEE Press, 2013.

[62] Elizabeth Whitworth and Robert Biddle. The social nature of agile teams. In *Agile conference (AGILE), 2007*, pages 26–36. IEEE, 2007.

[63] Olaf Zimmermann. Microservices tenets: agile approach to service development and deployment. *Computer Science-Research and Development*, 32(3):301–310, 2016.

Appendix A

Interview structure (Finnish)

```
TAUSTA

- Kerro roolistasi Kaiku Healthilla

SUUNNITTELU

 - Oliko projekti mielestäsi hyvin suunniteltu alunperin ja
    tuliko suunnitelmiin muutoksia kehityksen aikana
- Millaisia riskejä tunnistit projektissa teknisessä- ja
   bisnesmielessä?
        - Toteutuivatko riskit mielestäsi?
        - Tuliko yllätyksenä riskejä, joita ei oltu osattu
           ajatella?
        - Olivatko otetut riskit mielestäsi ottamisen arvoisia?

KEHITYS

- Miten olet osallistunut, olitko mukana alusta asti, mikä oli
   roolisi projektissa?
- Oliko ylimääräisiä prosesseja, puuttuiko jotain?
- Onko sinulle ollut yleistä että joudut aloittamaan uuden teht
   ävän edellisten ollessa kesken, onko tästä ollut haittaa?
- Olitko koko ajan perillä kehityksen tilasta?
- Ylimääräisen työn määrä:
        - Oliko eri asioiden odottelua, hyväksyntää, testausta?
- Dokumentaatio ja handoffit?
- Tuotteen tila:
```

```
        - Särkyvyys
        - Uudelleenkäytettävyys
        - Helpompi tehdä asioita väärin kuin oikein?
        - Turhaa kompleksisuutta?
        - Turhaa toistoa?
- Tekninen velka, kuinka minimoida tällaisessa projektissa?
- Mielipide testauksesta, oliko riittävää?
- Oliko projekti teknisessä mielessä onnistunut?
    - Olisitko tehnyt joitain teknisiä valintoja toisin?
    - Menikö tuotteen arkkitehtuuri huonompaan vai parempaan
      suuntaan?
    - Oletko kehittänyt uutta käyttöliittymää? Miten koet sen
      parissa työskentelemisen verrattuna vanhaan?
- Mielipiteesi microservice-arkkitehtuurista?
    - Kannattaako siihen suuntaan mennä, miten se soveltuisi
      meille?
- Tiimidynamiikka? olisiko jotain voinut parantaa, koetko
   tuotteen kehityksen motivoivana?

JULKAISU

- Oliko julkaisu mielestäsi onnistunut, miten määrittelet
   onnistuneen julkaisun?
    - Miksi? Mitä olisi voinut parantaa?
    - Onko uusi codebase mielestäsi tarpeeksi vakaa (
      toiminnallisuudet, luotettavuus)?

REFLEKTIO

- Oliko projektin aikataulu onnistunut?
- Oliko projektissa tarpeeksi kehitysresursseja?
- Oliko projekti manageroitu tarpeeksi hyvin?
- Oliko projekti bisnesmielessä merkittävä?
- Startup-kontekstissa, oliko projektissa joitain uniikkeja
   haasteita?

- Mielipiteesi yrityksen kehitysmenetelmistä?
    - Mitä kipupisteitä olet tunnistanut, onko kehitysideoita?
    - Toimivatko esim. Kanban ja dailyt?

- Oliko projekti kokonaisuutena onnistunut? miten määrittelisit
```

```
    projektin onnistumisen yleisesti?
- Mitä opit projektista, mikä on olennaista tämänkaltaisen
   projektin onnistumisessa?
```